CDCNET

MC68000™ Cross-Assembler

Reference

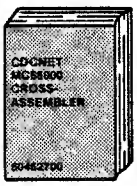
This product is intended for use only as described in this manual. Control Data cannot be responsible for the proper functioning of undescribed features or parameters.

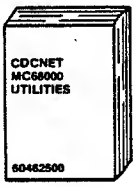
Publication Number 60462700

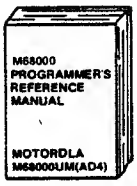
RELATED PUBLICATIONS

Background (Access as Needed):

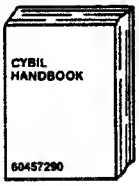
Software Development:

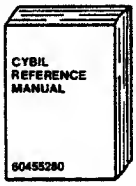


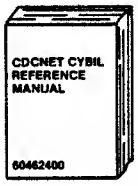




CYBIL References:







MANUAL HISTORY

This manual is Revision 01, printed 10/84. It is the Preliminary Release under NOS Version 2.

CONTENTS

ABOUT THIS MANUAL

This manual describes the CDCNET MC68000™ Cross-Assembler (Cross-Assembler), which runs under the Network Operating System (NOS) Version 2. These software products operate on CDC CYBER 70 Computer Systems Models 71, 72, 73, and 74, CDC® CYBER 170 Computer Systems, CDC CYBER 180 Computer Systems, and CDC 6000 Computer Systems. The Cross-Assembler is part of the CDCNET support tools product.

AUDIENCE FOR THIS MANUAL

This manual assumes you are an assembly language programmer who is developing software that will execute on the MC68000 microporocessor in CDCNET device interfaces (DIs). You need to be familiar with the Motorola MC68000 microprocessor, the NOS system commands, and CDC CYBER computer systems.

ORGANIZATION

This manual consists of six chapters. Chapter 1 describes the purpose of the CDCNET MC68000 Cross-Assembler. Chapter 2 describes Cross-Assembler program elements and coding rules. Chapters 3 and 4 describe the pseudo and machine instruction statements that the Cross-Assembler supports. Chapter 5 describes and illustrates macro capabilities, and Chapter 6 explains how to run the Cross-Assembler.

CONVENTIONS

The following conventions apply to user entry formats presented in this manual.

- Square brackets indicate optional constructs.
- Expr1 and expr2 indicate arithmetic expressions.
- Apostrophes or quotation marks indicate a sequence of ASCII characters.
- Number[s] indicates one or more numbers separated by commas.
- Expression[s] indicates one or more expressions separated by commmas.
- Symbol[s] indicates one or more symbols separated by commas.
- Ellipsis [. . .] indicates any number of repetitions.

INTRODUCTION 1

INTRODUCTION 1

This section describes the purpose of the CDCNET MC68000™ Cross-Assembler (Cross-Assembler), and describes the restrictions the Cross-Assembler places on the format of the source program.

PURPOSE

The Cross-Assembler is a software program that generates programs and code for loading into and executing on a MC68000-based device. The primary purpose of the Cross-Assembler is to aid in the development of software for the CDCNET device interface (CDI). The Cross-Assembler can also be used for development of software for other MC68000-based systems.

SOURCE PROGRAM RESTRICTIONS

The Cross-Assembler places the following restrictions on the format of the source program.

- A REPT pseudo instruction should not appear within an IF - IFEND block that is nested inside a MACRO definition. If this type of code sequence occurs inside the program, the results are unpredictable. To avoid this, you should always use .IF inside MACRO definitions.

- IF - ELSE - IFEND sequences within macros may only be used if the text within the IF structure does not contain any macro constructs. In other words, no parameter substitution, unique label generation, or local symbol substitution will take place within the IF - IFEND block. For this reason, you should always use .IF instead of IF within macros.

CROSS-ASSEMBLER PROGRAM ELEMENTS AND CODING RULES 2

This chapter describes program elements and coding rules, and specifies valid program formats for the Cross-Assembler.

CROSS-ASSEMBLER PROGRAM ELEMENTS AND CODING RULES 2

This chapter describes all the Cross-Assembler program elements and coding rules. This chapter also specifies which program formats will be accepted as valid input for the Cross-Assembler.

STRUCTURE OF A CROSS-ASSEMBLER SOURCE MODULE

A Cross-Assembler source module is the name given to an entity that the Cross-Assembler treats as one complete independent unit. The Cross-Assembler has the capability of processing several source modules in one run, but it effectively reinitializes itself at the beginning of each source module. Previous source modules have no effect on any subsequent source modules within the entire source module.

The following example shows how source modules can be grouped into a larger source module.

```
"68000" statement #1
. . . zero or more assembler instructions . . .
      END statement #1
"68000" statement #2
. . . zero or more assembler instructions . . .
      END statement #2
           .      .
           .      .
           .      .
"68000" statement #n
. . . zero or more assembler instructions . . .
     END statement #n
```

"68000 STATEMENT"

Purpose — The "68000" statement identifies the beginning of a new source module. This statement also indicates information pertaining to the current source module such as the object code module name, listing options, and assembly-control options.

Format — `"68000"  NAME  [options]`

Parameters — NAME specifies the current source module being assembled. This name also identifies the assembly object module. The source module name must obey NOS file name conventions. The maximum number of characters is seven, and all of these characters must be alphanumeric.

You can select none or all of the following options for the "68000" statement format. These options control the assembly and must be separated by one or more spaces.

Option	Description
NOLIST	Specifies that the source listing, except for lines that contain errors, will be suppressed. All LIST pseudo instructions within the source program are ignored.
LIST	Specifies that the source listing will contain one line for each line in the source file, and that macro expansion will not be shown on the listing. All NOLIST pseudo instructions in the source program are ignored.
EXPAND	Specifies that the source listing will contain both a list of all source lines entered and a list of all macro expansions. All LIST pseudo instructions in the source program are ignored.
NOCODE	Suppresses the generation of object code for this source module. This permits fast assembly so that the syntax of the source program may be checked quickly.
XREF	Generates a cross-reference listing for the current module. All symbols used within the program are listed in alphabetical order. Also, all lines from which each of the symbols was referenced are displayed.

Remarks

- The "68000" statement must be the first line of the source module.
- No comments or blank lines may precede the "68000" statement.
- Because the Cross-Assembler has the capability of assembling multiple source modules at a time, any line that follows the END statement must be the "68000" statement of the next source module.

SOURCE STATEMENT FORMAT RULES

Each source statement that the Cross-Assembler accepts is a line from a source module. The four possible types of source statements are as follows.

1. Microprocessor instruction statements
2. Cross-Assembler pseudo instruction statements
3. Macro call statements
4. An empty line or a comment line

Each source statement for microprocessor instruction statements, Cross-Assembler pseudo instruction statements, and macro call statements is divided into the following four fields.

1. Label field
2. Operation field
3. Operand field
4. Comment field

Each source statement may be up to 110 characters in length. Anything beyond this maximum will be truncated. There are no continuation lines in the language. The rules for forming each of these fields is described in their respective sections in this chapter.

Label Field

Labels can occur on all microprocessor instructions, all macro calls, and on some Cross-Assembler pseudo instructions. Every label must be unique within each given source module. The reason for this is that the assigned label identifies that particular statement and this label may be used as a reference point by other statements in the program. However, a macro name and a statement label may have the same name because the context of their uses will determine whether the name refers to a macro or a statement label.

The label field starts in column one of the source file and is terminated by a space or a colon. A label may contain any number of characters but only the first 15 are significant. The first character must be alphabetic. The remaining characters may be alphabetic, numeric, or underscores. Lowercase characters are not equivalent to their corresponding uppercase characters. For example, LaBel and LABEL are not the same symbol.

The only statements for which labels are mandatory are macro definitions and EQU and SET instructions. For all other statements, labels are optional.

Operation Field

The operation field contains a mnemonic code for a microprocessor instruction, a pseudo instruction, or a macro call. The pseudo instruction specifies the operation or function to be performed. This required field begins at the first non-space character after the label field, and is terminated by a space or a tab.

Operand Field

The operand field specifies values or locations required by the instruction in the operation field. The operand field begins at the first non-space character after the operation field. It is terminated by a space, tab, carriage return, or semicolon.

The operand may contain an expression consisting of a single symbolic term, or a combination of symbolic and numeric terms. The operand must be enclosed in parentheses, and joined by the expression operators described under Format of Valid Expressions in this chapter.

Comment Field

The comment field is optional. This field contains information that you believe is necessary to identify portions of the program. This information is not processed by the Cross-Assembler.

The delimiter for the comment field can be an asterisk in column one of a source statement. The delimiter can also be a semicolon, a tab, or a space that follows the operand field. A semicolon used in any column of the source statement will always start a comment except if the semicolon is part of an ASCII string.

FORMAT OF VALID EXPRESSIONS

Absolute and relocatable are the two basic classes of valid expressions that the Cross-Assembler's expression evaluator expects. The operands composing the expressions and the operators that are applied to these operands determine whether an expression can be absolute or relocatable. All possible operands and operators are described in the following sections of this chapter.

Symbols

The following symbols can be used in an expression.

- Label. This symbol appears in the label field of a machine instruction or a macro call. This symbol may also appear in one of the following Cross-Assembler pseudo instructions.

  - ASC
  - BIN
  - DEC
  - HEX
  - OCT

  All labels defined in the relocatable sections PROG, DATA, and COMN, are called relocatable. All labels defined in an ORG section are called absolute symbols.

- External symbol. This symbol appears in the operand field. This symbol is defined with the EXTERNAL pseudo instruction and is relocatable.

- Equated symbol. This symbol appears in the label field of an EQU or a SET pseudo instruction. This symbol is relocatable only if the expression to which it is equated is relocatable.

Numeric Constants

Numeric constants are either hexadecimal, decimal, octal, or binary numbers. The suffix attached to the number determines its form. B is attached to binary numbers; O or Q is attached to octal numbers; and H is attached to hexadecimal numbers. A number is assumed to be decimal if none of these four suffixes is attached to it.

Numeric constants are treated as 32-bit quantities and as absolute expressions. Numeric constants must lie within the range of -2,147,443,648 to 4,294,887,295.

String Constants

A string constant is a sequence of characters used in an expression and enclosed in either quotation marks or apostrophes. Like numeric constants, string constants are treated as 32-bit quantities and are limited to a maximum of four significant characters. If the string constant exceeds this length, only the first four characters are significant. All characters after the fourth are ignored. If the length of the string is less than four characters, the string is padded on the left with null characters (zero bytes). String constants are treated as absolute expressions. The value of a string constant is calculated by using the ASCII values of the characters within the string. This value is also affected by the current MASK values. For a detailed explanation of the MASK pseudo-instruction statement see chapter 3.

To use an apostrophe within a string, you must use double quotes as the string delimiters. Similarly, to use a double quote within a string, you must use apostrophes as the string delimiters.

Dollar Sign ($)

The value of the dollar sign ($) is the current value of the location counter for the current section. If the current section is PROG, DATA, or COMN, the dollar sign is a relocatable symbol. If the current section is ORG, the dollar sign is an absolute symbol.

Operators

The following list shows all of the operators that are accepted by the Cross-Assembler. This list also gives a brief explanation of the operation performed and the arguments expected. The operators are listed in descending order of precedence. The first three operators are unary and expect one argument following them. The remainder of the operators are binary and must occur between two expressions.

Operator	Meaning
+	Unary plus
-	Unary minus
.NT.	Logical one's complement
.SL.	Arithmetic left shift
.SR.	Arithmetic right shift
*	Multiplication
/	Integer division
+	Binary addition
-	Binary subtraction
.AN.	Logical bitwise AND
.OR.	Logical bitwise OR
.EQ.	Boolean comparatives - equal
.NE.	Not equal
.GT.	Greater than
.GE.	Greater than or equal
.LT.	Less than
.LE.	Less than or equal

VALID COMBINATIONS OF RELOCATABLE AND ABSOLUTE EXPRESSIONS

The value of an absolute expression is known at assembly time. The value of a relocatable expression, however, is not completely determined until after the program has been linked. A valid relocatable expression evaluates to an offset into one of the relocatable sections (PROG, DATA, COMN), or to an offset from an external label. At link time, the linker must add actual address values to these relocatable expressions depending on where the relocatable sections are to be loaded into memory.

In the following definitions of expressions, the type of a relocatable expression refers to the relocatable section or external symbol to which the expression corresponds. Two relocatable expressions are of the same type only if both expressions correspond to the same relocatable section or the same external symbol.

The following are definitions of the six possible valid combinations of relocatable and absolute expressions.

1. Any numeric constant, string constant, or absolute symbol, including the dollar sign in ORG sections, is an absolute expression.

2. Any relocatable or external symbol, including the dollar sign in relocatable sections, is a relocatable expression.

3. Parentheses can be used within expressions to force the order of operations. Parenthetic expressions always take the highest precedence.

4. If REL_EXPR1 and REL_EXPR2 are two relocatable expressions of the same type the following are all valid absolute expressions.

```
REL_EXPR1-REL_EXPR2
REL_EXPR1[OP]REL_EXPR2
     where [OP] is one of .EQ. .NE. .GT.
                          .GE. .LT. .LE.
```

   If REL_EXPR1 and REL_EXPR2 do not refer to the same relocatable section or external symbol, the assembly aborts and an error message is issued.

5. If ABS_EXPR is an absolute expression and REL_EXPR is a relocatable expression, the following are all valid relocatable expressions of the same type as REL_EXPR.

```
ABS_EXPR+REL_EXPR
REL_EXPR+ABS_EXPR
REL_EXPR-ABS_EXPR
```

6. If ABS_EXPR1 and ABS_EXPR2 are absolute expressions, the following are valid absolute expressions.

```
[unary operation]ABS_EXPR1
ABS_EXPR1[binary operation]ABS_EXPR2
```

An expression must be of one of the six types described above in order to be valid. However, the Cross-Assembler automatically converts relocatable expressions to absolute expressions when they are used with any operator except binary plus or minus. The Cross-Assembler does this by assuming that the relocatable part of the expression is zero. This provides the ability to convert relocatable expressions to absolute by multiplying them by the number one. For an example of this conversion see RELORG in chapter 5.

ADDRESSING MODES AND CROSS-ASSEMBLER SYNTAX

The addressing modes used by the Cross-Assembler are identical to those listed in the Motorola M68000 16/32-Bit Microprocessor Programmer's Reference Manual, available from the Motorola Corporation. The only difference is that the Cross-Assembler expects the use of square brackets instead of parentheses.

PSEUDO INSTRUCTION STATEMENTS 3

This chapter contains detailed descriptions of individual Cross-Assembler pseudo instruction statements.

PSEUDO INSTRUCTION STATEMENTS 3

This chapter contains detailed descriptions of individual Cross-Assembler pseudo instruction statements. These pseudo instructions provide information to the Cross-Assembler and provide the following capabilities.

- Listing control.
- Linkage control.
- Program section control.
- Data generation.
- Constant definition.
- Alteration of code generated by MC68000 machine instructions.

For a summary of these pseudo instruction statements see Appendix C.

ABS(OLUTE) SHORT, ABS(OLUTE) LONG — SET ADDRESSING MODE

The ABSOLUTE_SHORT and ABSOLUTE_LONG addressing pseudo instructions force short and long absolute addressing modes respectively. These pseudo instructions can be used anywhere in the program module and remain in effect until the next instruction of this type is encountered.

The syntax of the ABS(OLUTE)_SHORT and ABS(OLUTE)_LONG addressing pseudo instructions is as follows.

```
Label          Operation          Operand

[symbol]       ABS[OLUTE]_SHORT
               or
[symbol]       ABS[OLUTE]_LONG
```

The Cross-Assembler always defaults to the long absolute addressing mode when the Cross-Assembler encounters a forward reference to a program label, or when the label is external. The Cross-Assembler optimizes to the short absolute addressing mode when the label has been predefined, or the address is a numeric value. A short address must be less than or equal to 7FFF hexadecimal.

The following are some examples of the ABS_SHORT and ABS_LONG addressing modes.

```
LOC     OBJ.    CODE    LABEL   OPERATION       OPERAND

0000    33C4                    MOVE            D4,DEST
0002    0000    0290
                                ABS_SHORT
0006    31C4    0290            MOVE            D4,DEST
                                ABS_LONG
000A    33C4                    MOVE            D4,DEST
000C    0000    0290

0290                    DEST    DS.W            1
```

ASC[II] — DEFINE AN ASCII STRING

The ASC[II] pseudo instruction stores ASCII text in memory using quotation marks or apostrophes as delimiters. Whichever delimiter was used to initiate the string must also be used to terminate it.

The syntax of the ASC pseudo instruction is as follows.

```
Label          Operation      Operand

[symbol]       ASC[II]        string
```

The ASCII characters specified in the operand field can be in the form of any valid string expression. However, only one such ASCII string is permitted on the statement. The ASC instruction will always generate an even number of characters in the string. If you specify an odd number of characters, the assembler will add a space to the end of the string in order to force it to an even number of characters. If you want an odd number of characters in the string expression, use the DC.B instruction.

BIN(ARY), DEC(IMAL), HEX, OCT(AL) — DEFINE CONSTANT

The BIN[ARY], DEC[IMAL], HEX, and OCT[AL] pseudo instructions store data in memory in binary, octal, decimal, or hexadecimal format [respecively].

The syntax of the BIN[ARY], DEC[IMAL], HEX, and OCT[AL] pseudo instructions is as follows.

Label	Operation	Operand
[symbol]	BIN[ARY]	binary number[s]
	or	
[symbol]	DEC[IMAL]	decimal number[s]
	or	
[symbol]	HEX	hexadecimal number[s]
	or	
[symbol]	OCT[AL]	octal number[s]

The suffixes B, D, O, and H are implicitly assumed because of the operation code. These suffixes are not appended to the end of the numbers used in the operand field. If more than one operand is specified, each one must be separated from the others by commas. Each number specified in the operand field occupies 32 bits of memory. To generate numbers that are less than 32 bits in length, use the DC.B or DC.W instructions.

COMN, DATA, ORG, PROG — DESIGNATE MEMORY STORAGE AREA

The COMN, DATA, ORG, AND PROG pseudo instructions permit change from one code section to another.

The syntax of the COMN, DATA, ORG, and PROG pseudo instructions is as follows.

Label	Operation	Operand
	COMN	
	or	
	DATA	
	or	
	ORG	absolute address
	or	
	PROG	

Three program counters can be used to identify areas of relocatable code in addition to the program counters used for the absolute code sections. The relocatable areas are designated as data (DATA), program (PROG), and common (COMN). The absolute areas are controlled by the ORG instruction.

Although the PROG and DATA instructions are functionally identical, they have two different names to identify two separate relocatable memory areas. COMN allows construction of a common block of data similar to the common blocks of FORTRAN used by different program modules. All labels appearing in the PROG, DATA or COMN sections are treated as relocatable labels.

ORG is only used for absolute programming. ORG sets the contents of the location counter to the address entered in the operand field. The statement immediately following the ORG instruction will be located at the address specified. All labels appearing in an ORG section are treated as constants. The following sequences of code are considered identical by the assembler.

```
        ORG  100H                    ...
LABEL   ...                  LABEL   EQU  100H
```

The ORG instruction cannot alter the relocatable area counters associated with the DATA, PROG, and COMN instructions. This is because the relocatable area instructions do not contain operands, their associated counters start at zero, and their counters are initialized at linking time. However, the DS instruction may be used to set the offset into any of these relocatable sections. This simulates the execution of ORG instruction into the relocatable sections.

The default memory area (PROG) is used when constructing a source program. The DATA memory area might occupy another part of memory and can be used for storing data, tables, and instructions. The COMN pseudo instruction can be used to group information common to a number of program units. Assigning these types of items to a specific area in memory facilitates modification and referencing.

DC — DEFINE CONSTANT

The DC instruction stores a constant in memory starting with the current setting of the program counter.

The syntax of the DC pseudo instruction is as follows.

Label	Operation	Operand
[symbol]	DC.B	expression[s] or string
	or	
[symbol]	DC.W	expression[s]
	or	
[symbol]	DC.L	expression[s]

The DC instruction may contain more than one operand, but each must be separated by a comma. The operands can be symbols, expressions, or numbers that the Cross-Assembler can evaluate to numerical values. The constant will be aligned on a word boundary if word (W) or longword (L) is specified. If W or L is not specified, the constant is aligned to a byte boundary.

The DC.B instruction allows a string as an operand. However, a string can be the only operand specified for the instruction. This instruction also allows the generation of messages with an odd number of characters. For information on generating messages with an even number of characters, see the ASC[II] pseudo instruction in this chapter.

The label symbol is optional. When present, it is assigned the starting value of the location counter, and references the first constant stored by the instruction.

DS — RESERVE STORAGE

The DS instruction defines a block of memory.

The syntax of the DS pseudo instruction is as follows.

Label	Operation	Operand
[symbol]	DS.B	absolute expression
	or	
[symbol]	DS.W	absolute expression
	or	
[symbol]	DS.L	absolute expression

The value of the expression in the operand field specifies the number of bytes (B), words (W), or longwords (L) to be reserved. Any symbol appearing in the operand field must be defined before this instruction is encountered in the source file. If the value of the operand expression is zero, no memory will be reserved. If the optional label symbol is present, it will be assigned the current value of the location counter. If the value of the operand is less than zero, an error will occur.

The DS instruction increases the location counter by a specified value within the relocatable sections PROG, DATA or COMN. This cannot be done with ORG because this pseudo instruction causes a different section to be generated.

END — TERMINATE A PROGRAM UNIT

The END pseudo instruction signifies the logical end of a program unit.

The syntax of the END pseudo instruction is as follows.

Label	Operation	Operand
	END	[global label]

Each program unit must begin with a "68000" statement and terminate with an END statement. The optional label in the operand field specifies the starting address in memory for program execution. This address is also known as the transfer address. The label must be declared to be a global symbol in order to be used in this context. The label must also be defined in the module in which it is used as the transfer address.

EQU, SET — EQUATE SYMBOL TO A VALUE

The EQU and SET pseudo instructions establish a relationship between a symbol and an expression. The symbol in the label field acquires the same value as the expression in the operand field.

The syntax of the EQU and SET pseudo instructions is as follows.

Label	Operation	Operand
symbol	EQU	expression
	or	
symbol	SET	absolute expression

An EQU instruction equates external and equated symbols to either absolute or relocatable types. The SET instruction, however, can only be used with absolute expressions. A symbol in a SET instruction can be redefined, but a symbol in an EQU symbol cannot be redefined. For a complete discussion of possible expression formats see Format of Valid Expressions in chapter 3.

The following are some examples of the EQU and SET pseudo instructions.

```
Label       Operation     Operand

MPC_IBC     EQU           6
UACCESS     EQU           0C0H+(MPC_IBC)/2
DEBUG       SET           TRUE
DEBUG       SET           FALSE
            EXTERNAL      TABLE
SYMNAME     EQU           TABLE+5
```

EVEN — SET PROGRM COUNTER TO EVEN ADDRESS

```
Label       Operation     Operand

[symbol]    EVEN
```

The EVEN pseudo instruction effectively ensures that the program counter is aligned on a word boundary for the next instruction. If a label is present, it is assigned the address of the location counter before any alignment is performed.

EXPAND — ENABLE CROSS-REFERENCE LISTING

```
Label       Operation     Operand

            EXPAND
```

The EXPAND pseudo instruction can be used in the "68000" statement or as a stand-alone statement in the source module. If embedded in the source module, EXPAND will generate, within the output listing, all macro and data expansions that follow it. The EXPAND mode may be exited by embedding the LIST directive at any point following the EXPAND directive within the source.

EXTERNAL — IDENTIFY LABELS WHICH ARE DECLARED EXTERNALLY

```
Label       Operation     Operand

            EXTERNAL      symbol[s]
```

Symbols that are used in one program unit but are defined in a different program unit must be declared external with an EXTERNAL statement. When the Cross-Assembler processes the source statement, it puts information into the object text that describes the external symbols to the linker. At linkage time, the linker modifies the code so that all references to the external symbols contain the proper addresses. All external addresses must be 32 bits in length because of the CDC object text format.

GLB — DECLARE LABEL TO BE GLOBAL

Label	Operation	Operand
	GLB	symbol[s]

Symbols must be declared to be global in the program unit in which they are defined if these symbols are referenced by other program units. This is done through the use of the GLB statement. The Cross-Assembler indicates all symbols that were declared global in the object codes for the linker. At linkage time, the linker matches all external references with global symbols. All labels that are declared to be global must be relocatable labels. Labels that are used in ORG sections are treated as constants and must be declared via EQU statements in other modules.

The following are some examples of the GLB pseudo instruction.

Label	Operation	Operand
	GLB	KILL_MEMORY,INIT_MEM
	GLB	READ_CLOCK

IF, ELSE, IFEND, ENDIF — CONDITIONAL ASSEMBLY

The IF, ELSE, IFEND, and ENDIF pseudo instructions permit conditional assembly.

The syntax of the IF, ELSE, IFEND, and ENDIF pseudo instructions is as follows.

Label	Operation	Operand
	IF	absolute expression
	...	
	ELSE	
	...	
	IFEND or ENDIF	
	or	
	IF	absolute expression
	...	
	IFEND or ENDIF	

The IF pseudo instruction permits sections of code to be conditionally assembled. Sections of code are assembled or skipped as specified on the absolute expression. This expression is treated as a boolean value with either a true (non-zero) or a false (zero) value. The expression type must be absolute, and all symbols used in the expression must be previously defined in the source.

When the expression evaluates to a true condition, the code following the IF is assembled until an ELSE or an IFEND or ENDIF is encountered. If the expression evaluates to false, the ELSE part of the IF instruction is assembled until an IFEND or ENDIF is found.

The IFEND or ENDIF instructions are used to terminate the IF instruction. They must follow either the ELSE instruction or the IF instruction if no ELSE portion is desired.

Conditional IF instructions can be nested up to 20 levels deep. If the nesting levels exceed 20, an appropriate error message is issued. The occurrence of an ELSE, IFEND, or ENDIF instruction without a matching IF instruction also generates an error.

Whenever an IF-ELSE-IFEND sequence is used within a macro, no macro constructs can occur within the sequence. If this happens, none of the macro constructs are processed and error messages are issued. To avoid this, use .IF sequences in place of IF within macros.

The following are some examples of the IF, ELSE, IFEND, and ENDIF pseudo instructions.

```
Label       Operation     Operand

            IF            DEBUG
            JSR           WRITE_MESSAGE
            ASCII         "TRACE - Entering GETCHR"
            DC.B          0
            IFEND

            IF            BAUD.EQ.1200
            MOVEI         #DELAY_1200,D1
            ELSE
            MOVEI         #DELAY_300,D1
            IFEND
```

LIST, NOLIST — LISTING CONTROL OPTIONS

```
Label       Operation     Operand

            LIST
            or
            NOLIST
```

The LIST pseudo instruction can be used in the "68000" statement or as a stand-alone statement in the source program. If embedded in the source program, LIST generates one line of output for each line of source code that follows. All LIST instructions within the source program will be overridden if any list option was specified on the "68000" statement.

The NOLIST instruction is used in the "68000" statement or as a stand-alone statement in the source program. If embedded in the source program, it suppresses the output listing of all source statements that follow. If used in the "68000" statement, NOLIST suppresses all output listings except error messages.

MASK — SETUP STRING MASKING VALUES

The MASK pseudo instruction masks all ASCII strings defined in the source that follows the instruction.

The syntax of the MASK pseudo instruction is as follows.

```
Label       Operation     Operand

            MASK          expr1[,expr2]
```

The MASK pseudo instruction only affects ASCII strings. With each ASCII character this pseudo instruction produces a logical AND operation followed by a logical OR operation. The value of expr1 is the value with which each character will be ANDed. The value of expr2 is the value with which each character will be ORed. The value expr2 is optional.

The default condition at the beginning of each program unit is as follows.

```
AND value = 0FFH
OR  value = 0
```

NAME — ADD COMMENTS TO OBJECT MODULE

The NAME pseudo instruction adds comments to the object module for reference on the load map listing.

The syntax of the NAME pseudo instruction is as follows.

Label	Operation	Operand
	NAME	character string

The name string is limited to a maximum of 40 characters that can be any combination of alphabetic characters, numbers, or special characters.

REPT — REPEAT THE NEXT STATEMENT

The REPT pseudo instruction is used to repeat the next source statement any given number of times.

The syntax of the REPT pseudo instruction is as follows.

Label	Operation	Operand
	REPT	number

Only the following seven Cross-Assembler pseudo instructions make sense after a REPT statement. Unfavorable results may arise if any other pseudo instructions are attempted after a REPT.

1. ASC
2. BIN
3. DEC
4. HEX
5. OCT
6. DC
7. DS

A REPT pseudo instruction cannot be nested inside of an IF that is nested inside a macro definition. If this is done the results are unpredictable.

RORG, PC INDEP, NO RORG, PC DEP — DECLARE A POSITION INDEPENDENT PROGRAM SECTION

These pseudo instructions allow you to declare a position-independent program section.

The syntax of the RORG, PC_INDEP, NO_ORG, and PC_DEP pseudo instructions is as follows.

Label	Operation	Operand
[symbol]	RORG	
	or	
[symbol]	PC_INDEP	
	or	
[symbol]	NO_RORG	
	or	
[symbol]	PC_DEP	

The RORG, PC_INDEP, NO_ORG, and PC_DEP pseudo instructions should only be used in relocatable program modules. They should not be used with absolute program units containing ORG instructions (absolute program units). These pseudo instructions do not allow arguments and do not affect the current location counter. PC relative addressing modes are only generated after a RORG (PC_INDEP) pseudo instruction is encountered and will continue until the next NO_RORG (PC_DEP) pseudo instruction is encountered. These instructions do not have to be used in pairs. RORG and PC_INDEP are treated identically by the Cross-Assembler. NO_RORG and PC_DEP are also treated identically.

SKIP — PERFORM PAGE EJECT

The SKIP pseudo instruction causes the next line of output to be placed at the beginning of a new page.

The syntax of the SKIP pseudo instruction is as follows.

Label	Operation	Operand
	SKIP	

The SKIP pseudo instruction does not appear on the output listing unless an error occurs on this statement. If the NOLIST option has been selected, the SKIP pseudo instruction is ignored.

SPC — LEAVE BLANK LINES ON LISTING

The SPC pseudo instruction causes the Cross-Assembler to space downward (line feed) a specified number of lines.

Label	Operation	Operand
	SPC	[number]

The number of line feeds required is indicated in the operand field. If the operand field is left blank, the Cross-Assembler generates one blank line. The SPC instruction is not printed on the output listing unless an error occurs in the statement.

If the NOLIST option is in effect, the SPC instruction is ignored and does not affect the listing.

TITLE — DEFINE LISTING TITLE

The TITLE pseudo instruction initiates a page eject and causes the string expression to be printed as a title at the top of each page that follows.

The syntax of the TITLE pseudo instruction is as follows.

Label	Operation	Operand
	TITLE	name or string

The operand name or string operand (the title) may be a maximum of 70 characters in length and can be changed any number of times during the program. The TITLE instruction is not printed on the listing unless an error occurs on that statement.

If the NOLIST option has been selected, no page eject is performed.

The following are some examples of the TITLE pseudo instruction.

Label	Operation	Operand
	TITLE	CDNA Executive V1.0
	TITLE	Executive

MACHINE INSTRUCTION STATEMENTS 4

This chapter contains a table which summarizes machine instructions and syntax for the MC68000.

MACHINE INSTRUCTION STATEMENTS 4

This chapter contains a summary of machine instructions for the MC68000 and the syntax the Cross-Assembler expects for each instruction. See the Motorola M68000 16/32-Bit Microprocessor Programmer's Reference Manual for a complete description of the MC68000.

The following notation is used in this chapter.

- The Operand Assembler Syntax column describes the syntax the Cross-Assembler expects for the operands of each instruction. The following notation is used in this column.

<EA>	Any effective addressing mode.
Dy	Any data register (D0-D7).
Ay	Any address register (A0-A7).
#<DATA>	Any immediate data (for example, #5).
<LABEL>	Any program statement label.
Rx,Ry	Any machine registers.
#<VECTOR>	Any trap vector.

- The Condition Codes column describes the result of the instruction on each of the condition codes. XNZVC represent the extend, negative, zero, overflow and carry flags respectively. The following characters are used under each of these columns.

*	Set according to the result of operation.
-	Not affected by the operation.
0	Cleared.
1	Set.
u	Undefined.

- The Effective Modes column describes which effective addressing modes are valid for the given instruction. The following notation is used in this column.

M	The memory addressing modes can be used.
A	The alterable addressing modes can be used.
D	The data addressing modes can be used.
C	The control addressing modes can be used.
M^A	Memory alterable addressing modes are valid.
D^A	Data alterable addressing modes are valid.
C^A	Control alterable addressing modes are valid.
ALL	All addressing modes are valid.
A-	All alterable addressing modes are valid unless the operand size is byte, in which case the address register direct mode is not allowed.

Table 4-1. Machine Instruction Instruction Statements

Instruc. Mnemonic	Perm. Ext.	Operand Assembler Syntax	Cond. Codes XNZVC	<EA> Modes	Description
ABCD	B	Dy,Dx -[Ay],-[Ax]	*u*u*		Add decimal with extend.
ADD	B W L	<EA>,Dy Dy,<EA>	****)	ALL M^A	Add binary.
AND	B W L	<EA>,Dy	-----	ALL	Add address.
ADDI	B W L	#<DATA>,<EA>	*****	D^A	Add immediate.
ADDQ	B W L	#<DATA>,<EA>	*****	A-	Add quick.
ADDX	B W L	Dy,Dx -[Ay],-[Ax]	*****		Add extended.
AND	B W L	<EA>,Dy Dy,<EA>	-**00	D^A M^A	AND logical.
ANDI	B W L	#<DATA>,<EA> #<DATA>,SR	-**00	D^A	AND immediate.
ASL, ASR	B W L W	Dy,Dx #<DATA>,Dy <EA>	*****		Arithmetic shift.
Bcc	S L	<LABEL>	-----		Branch conditionally.
BCHG	B L	Dy,<EA> #<DATA>,<EA>	--*--	D^A D^A	Test a bit and change.
BCLR	B L	Dy,<EA> #<DATA>,<EA>	--*--	D^A D^A	Test a bit and clear.
BRA	S L	<LABEL>	-----		Branch unconditionally.
BSET	B L	Dy,<EA> #<DATA>,<EA>	--*--	D^A D^A	Test a bit and set.
BSR	S L	<LABEL>	-----		Branch to subroutine.
BTST	B L	Dy,<EA> #<DATA>,<EA>	--*--	D D	Test a bit.
CHK	W	<EA>,Dy	-*uuu	D	Check against bounds.
CLR	B W L	<EA>	-0100	D^A	Clear an operand.

(Continued)

Table 4-1. Machine Instruction Statements (Continued)

Instruc. Mnemonic	Perm. Ext.	Operand Assembler Syntax	Cond. Codes XNZVC	<EA> Modes	Description
CMP	B W L	<EA>,Dy	-****	ALL	Compare.
CMPA	W L	<EA>,Ay	-****	ALL	Compare address.
CMPI	B W L	#<DATA>,<EA>	-****	D^A	Compare immediate.
CMPM	B W L	[Ay]+,[Ax]+	-****		Compare memory.
DBcc	W	Dy,<LABEL>	-----		Test, decrement and Bcc.
DIVS	W	<EA>,Dy	-***0	D	Divide signed.
DIVU	W	<EA>,Dy	-***0	D	Divide unsigned.
EOR	B W L	Dy,<EA>	-**00	D^A	Exclusive OR logical.
EORI	B W L	#<DATA>,<EA> #<DATA>,SR	-**00	D^A	Exclusive OR immediate.
EXG	L	Ry,Rx	-----		Exchange two registers.
EXT	W L	Dy	-**00		Sign extend register.
JMP		<EA>	-----	C	Jump.
JSR		<EA>	-----	C	Jump to subroutine.
LEA	L	<EA>,Ay	-----	C	Load effective address.
LINK		Ay,#<DATA>	-----		Link and allocate.
LSL, LSR	B W L	Dy,Dx #<DATA>,Dy	***0*		Logical shift.
	W	<EA>		M^A	
MOVE	B W L	<EA>,<EA>	-**00	ALL D^A	Move data. Data alt. destination.
MOVE	W	<EA>,CCR	*****	D	Move to CCR.
MOVE	W	<EA>,SR	*****	D	Move to SR.
MOVE	W	SR,<EA>	-----	D^A	Move from SR.
MOVE	L	USP,Ay Ay,USP	-----		Move user stack pointer.

(Continued)

Table 4-1 Machine Instruction Statements (Continued)

Instruc. Mnemonic	Perm. Ext.	Operand Assembler Syntax	Cond. Codes XNZVC	<EA> Modes	Description
MOVEA	W L	<EA>,Ay	-----	ALL	Move address.
MOVEM	W L	RL,<EA> RL,-[Ay] <EA>,RL [Ay]+,RL	-----	C^A C	Move multiple registers.
MOVEP	W L	Dy,d[Ax] d[Ay],Dx	-----		Move peripheral data.
MOVEQ	L	#<DATA>,Dy	-**00		Move quick.
MULS	W	<EA>,Dy	-**00	D	Multiply signed.
MULU	W	<EA>,Dy	-**00	D	Multiply unsigned.
NBCD	B	<EA>	*u*u*	D^A	Negate decimal, extend.
NEG	B W L	<EA>	*****	D^A	Negate.
NEGX	B W L	<EA>	*****	D^A	Negate with extend.
NOP			-----		No operation.
NOT	B W L	<EA>	-**00	D^A	Logical compliment.
OR	B W L	<EA>,Dy Dy,<EA>	-**00	D M^A	Inclusive OR logical.
ORI	B W L	#<DATA>,<EA> #<DATA>,SR	-**00	D^A	Inclusive OR immediate.
PEA	L	<EA>	-----	C	Push effective address.
RESET			-----		Reset external devices.
ROL, ROR	B W L W	Dy,Dx #<DATA>,Dy <EA>	-**0*	 M^A	Rotate.
ROXL,ROXR	B W L W	Dy,Dx #<DATA>,Dy <EA>	***0*	 M^A	Rotate with extend.
RTE			*****		Return from exception.
RTR			*****		Return and restore CCR.

(Continued)

Table 4-1. Machine Instruction Statements (Continued)

Instruc. Mnemonic	Perm. Ext.	Operand Assembler Syntax	Cond. Codes XNZVC	<EA> Modes	Description
RTS			-----		Return from subroutine.
SBCD	B	Dy,Dx -[Ay],-[Ax]	*u*u*		Subtract decimal with extend.
Scc	B	<EA>	-----	D^A	Set conditionally.
STOP		#<DATA>	*****		Load SR and STOP.
SUB	B W L	<EA>,Dy Dy,<EA>	*****	ALL M^A	Subtract binary.
SUBA	W L	<EA>,Ay	-----	ALL	Subtract address.
SUBI	B W L	#<DATA>,<EA>	*****	D^A	Subtract immediate.
SUBQ	B W L	#<DATA>,<EA>	*****	A	Subtract quick.
SUBX	B W L	Dy,Dx -[Ay],-[Ax]	*****		Subtract with extend.
SWAP	W	Dy	-**00		Swap register halves.
TAS	B	<EA>	-**00	D^A	Test and set.
TRAP		#<VECTOR>	-----		Trap.
TRAPV			-----		Trap on overflow.
TST	B W L	<EA>	-**00	D^A	Test an operand.
UNLK		Ay	-----		Unlink.

MACROS 5

This chapter describes macro constructs and provides examples of macro capabilities on the Cross-Assembler.

MACROS 5

This chapter describes the use and definition of macros on the Cross-Assembler. In the beginning of this chapter all macro constructs are described in detail. At the end of this chapter, macro examples illustrate many of the available capabilities.

MACRO FORMAT

Macro definitions and macro calls (or expansions) are the two major classes of macro usage with the Cross-Assembler.

The general format of a macro definition is as follows.

```
name      MACRO          parameter_list
          macro_body
          MEND
```

Any number or type of instructions can be used within the macro body except nested macro definitions. For this reason, an MEND instruction must be encountered after a MACRO statement before the next MACRO instruction appears. Also, the use of the IF-ELSE-IFEND sequence within macros is not recommended. For further information on this pseudo instruction sequence, see IF, ELSE, IFEND, and ENDIF in chapter 3.

The macro name becomes the symbol that identifies that particular macro instruction. This name must be unique. The macro name cannot be the name of any previously defined macro, a pseudo instruction, or MC68000 machine instruction. Macro names must begin with an alphabetic character and may consist of any number of alphanumeric characters. However, only the first 15 characters are significant.

The parameter list consists of 0 to 34 formal parameters separated by commas. The format of each parameter is described in detail in Optional Parameters in this chapter.

OPTIONAL PARAMETERS

The parameters inside of a macro definition are referred to as formal parameters. The parameters that are specified in the macro call are referred to as actual parameters.

The parameter list in the MACRO statement is optional. If the parameter list is included it must consist of one or more formal parameters separated by commas. Each formal parameter must begin with an ampersand (&) followed by an alphabetic character followed by 0 to 14 alphanumeric characters. A maximum of 34 formal parameters may be specified. It is impossible to have more than 34 formal parameters because of the 110-character line length restriction.

Actual parameters appear in the operand field of the macro call statement and are separated by commas. An actual parameter may consist of any sequence of characters. However, if the actual parameter contains any spaces or commas, it must be enclosed in quotation marks or apostrophes. There is a one-to-one correspondence between the actual and formal parameters. The first formal parameter is replaced by the first actual parameter, the second formal parameter is replaced by the second actual parameter, and so on.

Parameter substitution is a textual replacement of the actual parameters for the formal parameters. Wherever the formal parameter occurs in the macro body, it is replaced by the exact sequence of characters specified in the corresponding actual parameter. For example, if the actual parameter is null, the formal parameter is replaced with the null string.

UNIQUE LABEL GENERATION

The Cross-Assembler provides the ability to generate unique label names within macros. This capability prevents duplication of labels when the macro is expanded more than once. The macro unique label construct consists of four consecutive ampersands (&&&&). Wherever this construct is encountered within a macro, the construct is replaced by a four-digit decimal number. This decimal number is padded on the left with zeros if its value is less than 1000. The unique label effect is obtained because this four-digit number is set to 0000 at the beginning of assembly and is incremented by one each time a macro is expanded. All occurrences of &&&& within a single macro expansion are replaced by the same four-digit code.

MACRO CONDITIONAL ASSEMBLY

The Cross-Assembler provides a set of four pseudo instructions that allow for looping and conditional assembly within macros. This capability is similar to the IF - ELSE - IFEND facility used outside of macros. This set of pseudo instructions consists of .IF, .SET, .GOTO, and .NOP. These pseudo instructions are described in detail in the following sections.

.IF Instruction

The .IF instruction provides the ability to conditionally branch to any line within the macro. The syntax of this instruction is as follows.

```
[local symbol]    .IF    expression    label
```

The value of expression must be absolute. If the value of expression is true (not zero), the Cross-Assembler will jump to the statement with label in its label field. In other words, the Cross-Assembler will either back up or skip forward in the source file until it finds the required line. If the value of expression is false, the assembler continues processing at the next line following the .IF statement.

Label must be a macro local symbol within the same macro as the .IF instruction.

The use of IF - ELSE - IFEND sequences within macros should be avoided. The .IF instruction should always be used for any conditional assembly within macros.

.SET Instruction

The .SET instruction is similar to the SET instruction used outside of macros. The .SET instruction allows macro local symbols to be equated to an absolute value. The format of this instruction is as follows.

```
label    .SET    absolute_expression
```

In the .SET instruction label becomes a macro local symbol that is equated to the value of the absolute_expression. The most common use of this facility is for loop counters. Each time through a loop the value of the local symbol can be decremented by one, and the loop can be executed repeatedly until the value of the symbol goes to zero.

.GOTO Instruction

The .GOTO instruction is used for unconditional branching within macros. This instruction is useful for creating looping constructs, or for skipping over sections of code. The format of this instruction is as follows.

```
.GOTO     label
```

.NOP Instruction

The .NOP instruction is only used to define macro local symbols. This instruction is commonly used as the target of a conditional or unconditional branch (.IF or .GOTO). The format of this instruction is as follows.

```
label     .NOP
```

MACRO LOCAL SYMBOLS

Macro local symbols are symbols that are only recognized within the given macro. When the macro is being expanded, a separate symbol table is created that contains all of the local symbols for this macro. At the end of the expansion of the macro, this separate symbol table is destroyed so that the local symbols are no longer known. A local symbol is defined as any label that occurs on an IF, .SET, or .NOP instruction. Local symbols are convenient for use as loop counters and labels for flow control. A maximum of 20 local symbols may occur within any macro definition.

CHECKING AND INDEXING PARAMETERS

There is a one-to-one correspondence between formal and actual parameters. Sometimes, however, it is more convenient to step through each of the actual parameters in the list in order instead of accessing the parameter by name. This is possible through the use of the parameter indexing operator &&. The two ampersands must be followed by a macro local symbol that appeared on a .SET instruction. The value of this local symbol indicates which actual parameter is intended. For example, if the value of a local symbol named COUNT is 1, the construct &&COUNT refers to the first actual parameter in the parameter list. If the value of COUNT is 7, &&COUNT refers to the seventh parameter in the list. This capability is typically used within a loop as follows.

```
ASC2    MACRO
COUNT   .SET    1
TOP     .IF     "&&COUNT".EQ.""     DONE
        ASCII   "&&COUNT"
COUNT   .SET    COUNT+1
        .GOTO   TOP
DONE    .NOP
        MEND
```

This example will loop through all parameters in the parameter list and define them as ASCII strings. Any number of parameters may be specified. The indexing operator is useful in defining macros that can handle a variable number of parameters. The third line in the example above shows a test to see if the parameter is null. This is a common method of determining when the end of the parameter list has been encountered. This is referred to as parameter checking.

MACRO EXAMPLES

This section includes samples of Cross-Assembler macro instruction usage. Each sample illustrates only one of many possible ways to define the given macro. These macro examples show only some of the many macro capabilities that the Cross-Assembler provides.

String Definition Macro

This macro permits the definition of strings so that the string length is contained in the first byte of the string. This is a common string format used in most versions of BASIC and in versions of Pascal that have extended the language to include strings.

In the following example, the unique label generation operator has been used to determine the length of the string.

```
STRING    MACRO   &P
          DC.B    LEN_&&&&
STRT&&&&  DC.B    &P
LEN_&&&&  EQU     $-STRT&&&&
          MEND
```

Recursive Factorial Macro

The following example illustrates the use of recursion in macros. This sample macro computes the factorial of the input parameter, and leaves the result in the global variable FACTVALUE.

```
FACTORIAL MACRO     &n
          .IF       &n.LE.1   FACT1
          FACTORIAL &n-1
FACTVALUE SET       (&n)*FACTVALUE
          .GOTO     FACTEND
FACT1     .NOP
FACTVALUE SET       1
FACTEND   .NOP
          MEND
```

Simulate Org in Relocatable Sections

This macro simulates the operation of ORG in the relocatable (PROG, DATA, COMN) sections of the program. This macro also illustrates the Cross-Assembler's ability to convert relocatable expressions to absolute by multiplying the expressions by 1. This conversion is necessary because subtracting a relocatable value from an absolute value is not valid.

```
RELORG    MACRO     &SECTION,&OFFSET
          &SECTION
          .IF       &OFFSET.GE.$   NOERROR
          ERROR     "Invalid offset specified"
          .GOTO     REND
NOERROR   .NOP
          DS.B      &OFFSET-1*$
REND           .NOP
          MEND
```

Calculate Maximum Value of Given Number Set

This macro calculates the maximum of a given set of numbers. The following macro illustrates the application of the macro parameter indexing operator to handle a variable number of parameters. This example also shows the use of macro conditional instructions to perform looping. These instructions assume that all of the values are non-negative, and leaves the resulting maximum value in the global variable MAXVALUE.

```
MAX       MACRO
MAXVALUE  SET       0
LPCNT     .SET      1
LOOP      .IF       '&&LPCNT'.EQ.''     MAXEND
          .IF       &&LPCNT.LE.MAXVALUE NEXT
MAXVALUE  SET       &&LPCNT
NEXT      .NOP
LPCNT     .SET      LPCNT+1
          .GOTO     LOOP
MAXEND    .NOP
          MEND
```

RUNNING THE CROSS-ASSEMBLER 6

This chapter describes the procedure call, the parameter format, and valid keywords for running the Cross-Assembler.

RUNNING THE CROSS-ASSEMBLER 6

The Cross-Assembler is available through the SES procedure ASM68K. The procedure call for the Cross-Assembler is as follows.

```
SES.ASM68K, parameters.
```

The parameters have the format keyword=value, keyword=(value), or keyword. Parameters are separated from each other by a comma. The following keywords are valid.

Keyword	Abbreviation	Description
INPUT=filename	I	The name of the file to be assembled. If this parameter is omitted, the Cross-Assembler assumes that input is on a local file named INPUT.
BINARY=filename	B	The name of the file to which the object code generated by the Cross-Assembler is to be written. This file is not rewound by the ASM68K procedure. If this parameter is omitted, the object code is written to a local file named LGO.
LISTING=filename	L	The name of the file to which the Cross-Assembler source listing is written. This file is not rewound by the ASM68K procedure. If this parameter is omitted, the listing is written to a local file named LISTING.
XREF	X	If this keyword is present, a symbol cross-reference table is included in the Cross-Assembler listing. If this keyword is omitted, no cross-reference table is generated.
DEBUG	D	If this keyword is present, debug symbol tables are included in the object code file. If this keyword is omitted, no debug symbol tables are generated.
COLUMNS: (left, right)	COLS or COL	The column numbers of the left and right margins for the Cross-Assembler source listing. If this parameter is omitted, the values (1,90) are used.
MSG		If this keyword is specified, informative messages are displayed at the terminal while ASM68K is executing.
NOMSG		If this keyword is specified, informative messages are not displayed at the terminal while ASM68K is executing. If this keyword is omitted, informative messages are displayed.

APPENDIXES

CROSS-ASSEMBLER ERROR MESSAGES A

This appendix contains an alphabetized list of Cross-Assembler error messages and an explanation for each error.

Message	Significance
A register required	This statement did not include the required A register as an operand.
argument subfield must be a symbolic name	The operand must be a symbol.
colon not after label	A colon was encountered which did not immediately follow a label and was not within a string.
digit inconsistent with base	An illegal digit was used in a number. This digit is illegal because of the base of the constant. For example, the digit 2 may not occur in a binary constant.
displacement value is out of range	The specified displacement will not fit in the field provided.
division by zero attempted	The Cross-Assembler will not allow division by zero in an expression.
expecting end of statement	Additional information was found on a statement line when none was expected. This usually occurs on statements which require no operands and have a comment which was used without a leading semicolon.
expecting expression	An expression was expected as an operand but none was found.
expecting formal parameter	The instruction following the MACRO pseudo instruction must be a formal parameter beginning with an ampersand (&).
expecting macro conditional after dot	A period which was not followed by a macro conditional (.IF, .GOTO, .SET, .NOP) was found in the operation field.
expecting right bracket	A left bracket was encountered for which the corresponding right bracket was not found. This could occur because the bracket is not there, or because the construct inside of the brackets is not a legal mode recognized by the Cross-Assembler.
expecting right parenthesis	A left parenthesis was encountered for which the corresponding right parenthesis was not found.

expecting "68000"	This statement must contain "68000" beginning in column 1 because the first statement of each module must be a "68000" statement.
expression must be absolute	A relocatable expression was encountered where an absolute expression was required.
illegal expression	An illegal expression was encountered by the expression evaluator. Different relocatable values were probably improperly mixed.
illegal or non-graphic character	This statement contains a character that is not recognized by the Cross-Assembler. Retype the entire line.
illegal register list	An illegal register list was encountered on a MOVEM instruction. A specification probably indicates the second register to be lower than the first. An example is D7-D3.
internal static table error	Something is internally wrong with the Cross-Assembler.
invalid addressing mode	The addressing mode specified for this statement is not one of the permissible modes for this instruction.
label ignored	The label will be ignored.
invalid character in parameter	Parameters may consist only of letters or digits and must begin with an ampersand (&).
invalid extension	The extension used is not valid for this operation.
invalid index register	An illegal register value was specified. Legal index registers are D0-D7 and A0-A7.
invalid macro name	A macro name must consist of an alphabetic character followed by any number of alphanumeric characters. In addition, a macro name cannot be the same as a machine instruction or a Cross-Assembler pseudo instruction.
invalid operator in expression	The expression contains an operator which the Cross-Assembler does not recognize.
label not a symbolic name	The label beginning in column 1 does not begin with an alphabetic character.
labels and operands ignored	Labels and operands will be ignored.
maximum IF nesting level exceeded	This IF statement causes the IF nesting level to exceed 20 levels.
maximum MACRO expansion nesting level	The macro nesting level has exceeded 100. The macro is probably executing an infinite recursive invocation.

maximum MACRO loop count exceeded	The current macro has gone through a loop more than 5000 times. A sequence of statements formed by .GOTO or .IF statements inside of the macro is probably executing an infinite loop.
missing IFEND statement	The END statement was encountered or the end of file was encountered where an an IFEND was expected.
missing MEND statement	The END statement or the end of file was encountered within a macro definition.
nested macro definition	A MACRO statement was encountered inside a macro definition. Nested macro definitions are not allowed.
operation prohibits base specification	The constants used with the pseudo instructions BIN, DEC, OCT and HEX cannot contain a trailing base specification.
operand type invalid	This type of operand is not valid for the given instruction.
operation subfield not a symbolic name	The operation field must contain a symbolic name.
PC or A register required	The addressing mode used is valid only if the program counter or one of the address registers is used. For example, there is no such addressing mode as [D5]+.
register required	The register required by the instruction was not specified.
required operand missing	The operand which the given instruction requires is missing.
statement label is not unique	The label appearing on this statement has already been defined in the source.
statement label required	This statement requires a label but none was found.
statement is valid only within an IF	An ELSE or an IFEND was encountered that was not preceded by a matching IF statement.
statement is valid only within a macro	An MEND or a macro conditional statement was encountered while outside a macro definition.
string too long	The string goes beyond the end of the physical source line. The terminating delimiter of a string was probably missing.
symbol must be global	Because the symbol appearing on the END statement is used as a transfer symbol, this symbol must be declared to be global.
syntax error	A syntax error was encountered.

syntax error in operation field	The operation field cannot be recognized as a valid symbol by the Cross-Assembler.
too many local symbols	The macro symbol table has overflowed. The maximum number of local symbols that can be used within a single macro definition is 35.
too many statement labels	A label was encountered that was followed by a comma instead of a space or a colon.
too many parameters	More than 34 formal parameters appear in the macro definition. This error should not occur because of the 110-character line limit.
undefined operation "..."	The symbol in the operation field is not recognized as a hardware instruction, pseudo instruction, or macro.
undefined symbolic name "..."	The name "..." is not defined in the source module.
value out of range	The specified value cannot be represented with the number of bits contained in the field.

CROSS-ASSEMBLER OBJECT FILE FORMAT B

The Cross-Assembler produces object code in the CDC Object Text Format. Different portions of the code generated for a program go into different sections of the object text. The linker allows the user complete control over the locations in memory into which each of these sections will be loaded. These locations are specified through the use of the Linker Parameter File (see the CDCNET MC68000 Utilities Manual for details). The following section names are used by the Cross-Assembler.

CODE	Everything that occurs in the relocatable PROG section in the source module will be placed in a CDC object text section named CODE.
COMN	Everything that occurs in the relocatable COMN section in the source module will be placed in a CDC object text section named COMN.
""	Everything that occurs in the relocatable DATA section in the source module will be placed in a CDC object text section with a null name.
ALS$..	Everything that occurs in the absolute ORG sections in the source module will be placed in a CDC object text section with the name ALS$xxxxxxxx, where xxxxxxxx is the address specified on the ORG statement and represented in hexadecimal notation.

SUMMARY OF PSEUDO INSTRUCTION STATEMENTS C

This appendix contains an alphabetized list of pseudo instruction statements and a description of the functions each statement performs.

Pseudo Instruction	Function Performed
ABS[OLUTE]_SHORT	Cause absolute addresses to default to 16 bits.
ABS[OLUTE]_LONG	Cause absolute addresses to default to 32 bits.
ASC[II]	Store data in memory using ASCII format.
BIN[ARY]	Store data in memory using binary format.
DEC[IMAL]	Store data in memory using decimal format.
HEX	Store data in memory using hexadecimal format.
OCT[AL]	Store data in memory using octal format.
DC.B, DC.W, DC.L	Store data in memory in any numeric format.
COMN	Following text assembles into the common area.
DATA	Following text assembles into data area.
ORG	Following text assembles into absolute memory.
PROG	Following text assembles into program area.
DS.B, DS.W, DS.L	Reserve storage.
END	Signifies the logical end of program unit.
EQU	Equate symbol to absolute or relocatable value.
SET	Equate symbol to absolute value only.
EVEN	Align program counter to word boundary.
EXPAND	Cause macro expansions to be shown on listing.
EXTERNAL	Declare symbols to be external to program unit.
GLB	Declare symbols to be globally defined.
IF, ELSE, IFEND	Define conditional code sequences.
LIST	Show only source lines on listing.
NOLIST	Suppress all lines except errors on listing.
MACRO, MEND	Define a macro instruction.
MASK	Define AND/OR values for ASCII mask.
NAME	Add comments to object module.
REPT	Repeat next source statement given number of times.
RORG, PC_INDEP	Change default to PC relative addressing.
NO_RORG, PC_DEP	Change default back to absolute addressing.
SKIP	Perform a page eject on listing.
SPC	Space downward a given number of lines on listing.
TITLE	Cause page eject and print title on every page.

COMMENT SHEET

MANUAL TITLE: CDCNET MC68000 Cross-Assembler

PUBLICATION NO.: 60462700 REVISION: 01

NAME:_____

COMPANY:_____

STREET ADDRESS:_____

CITY:_____ STATE:_____ ZIP CODE:_____

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments below (please include page number references).

☐ Please Reply ☐ No Reply Necessary

CUT ALONG LINE

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division
ARH219
4201 North Lexington Avenue
Saint Paul, Minnesota 55112

CUT ALONG LINE

FOLD FOLD